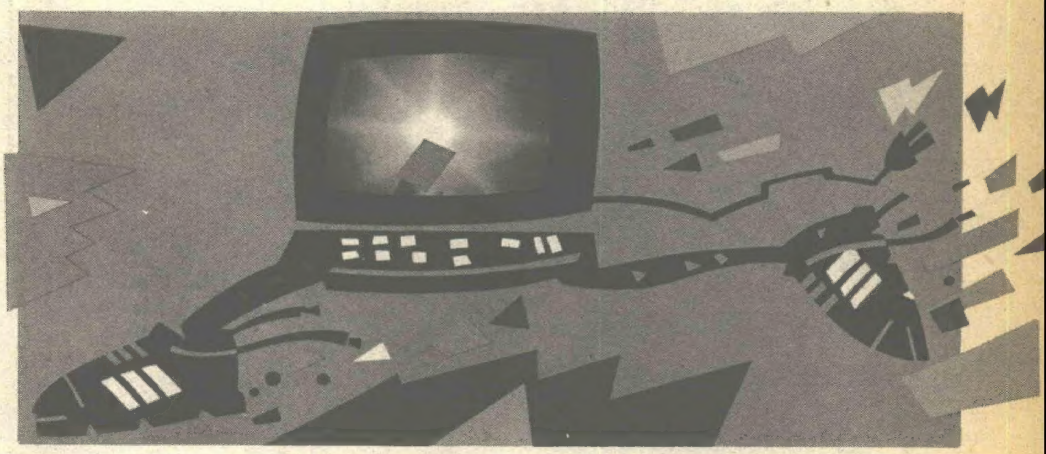
ซอฟต์แวร์
ลออ เพิ่มสมบัติ

เรียนรู้การใช้งานภาษา C++ ด้วยตนเอง : การใช้คลาสใน C++

ศึกษาเทคนิควิธีการใช้งานคลาสในภาษา c++ ด้วยตัวคุณเอง



ถ้าผมฝากเงินกับธนาคาร สิ่งที่ผมสนใจก็คือ เงินปัจจุบันที่มีอยู่ (current balance) กับอัตราดอกเบี้ย (rate) ซึ่งทุกธนาคารก็คงจะรักษาตัวแปร 2 ชนิดนี้คือ current balance และ interest rate (เป็นข้อมูลชนิดทศนิยม) อาจจะเป็นวัน เดือน ปี (คิดระยะเวลาเป็นสำคัญ) เมื่อประกาศใช้ใน C++ จะเป็น

```
double balance; // current balance
double rate; // decimal period interest
             // rate
```

ตัวแปรเหล่านี้จะเรียกว่า instance variable มีคีย์เวิร์ดสำหรับการประกาศคลาสคือ struct กับ class ใน 2 คำนี้ c++ จะใช้ class มากกว่า แต่เพื่อจะโยงมาจากภาษาซีจึงต้องใช้ struct ก่อน

คลาสที่ประกาศโดยใช้ struct

ในภาษาซีตัวแปรหลายตัวที่ต่างชนิดกันจะเก็บไว้ใน struct (ปาสคาลคือเรคอร์ด) เราจะลองประกาศบัญชีเงินฝากธนาคารโดยใช้คำว่า struct

```
struct account{
    double balance;
    double rate;
};
```

จะเรียกตัวแปรที่อยู่ภายใน struct ว่าสมาชิก ในที่นี้ account มีสมาชิก 2 ตัวคือ balance กับ rate

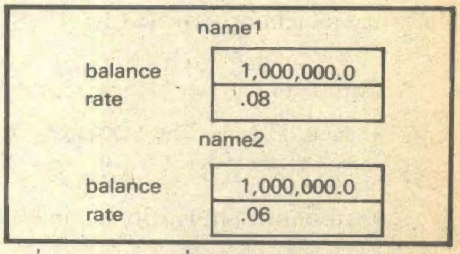
เราจะใช้ชื่อของ struct (ในที่นี้คือ account) ประกาศการใช้ตัวแปรใหม่นอกคลาสได้อีกด้วย

```
struct account name1;
struct account name2;
```

ใน c++ ไม่ต้องเขียนคำว่า struct อีกก็ได้ เช่น

```
account name1;
account name2;
```

จากรูปที่ 1 ตัวแปรที่อยู่ในคลาสมี 2 ตัวคือ balance และ rate ใน



รูปที่ 1 แสดงตัวแปรที่อยู่ในคลาส

c++ จัดให้มีโอเปอเรเตอร์ . (เรียกว่าจุด) ในการอ้างถึงสมาชิกของคลาสกับวัตถุของคลาส (อาทิเช่น name1, name2) ดังนั้น name1.balance จะหมายถึง ยอดเงินในบัญชีของ name1 ส่วน name2.rate หมายถึง อัตราดอกเบี้ยของบัญชี name2 (โอเปอเรเตอร์ . จะทำจากซ้ายไปขวาและมีลำดับความสำคัญสูงกว่าโอเปอเรเตอร์ตัวอื่น ๆ)

```
name1.balance = 10,000.0
```

หมายถึง name1 มีเงินฝากใน

บัญชีขณะนี้ 10,000 บาท

```
name2.balance = 1,000.0
```

หมายถึง name1 มีเงินฝากในบัญชีขณะนี้ 1,000 บาทเหมือนกัน

ถ้อยแถลง

```
if (name2.balance > = 5,000.0)
name2.balance - = 5,000.0;
```

หมายถึงถอนเงินจากบัญชี name2 5,000 บาท ถ้ามีเงินในบัญชีเพียงพอ

เมื่อเอา struct เข้าช่วยในการประกาศ ทำให้เราประกาศวัตถุ (บัญชีเงินฝาก) และใช้ชื่อของวัตถุเข้าถึงตัวแปรภายในคลาสได้ แต่การกระทำเช่นนี้ไม่ได้ซ่อนข้อมูลเอาไว้เลย ซึ่งผิดหลักของการเขียนโปรแกรมเชิงวัตถุที่มีการ encapsulate ดังนั้นถ้าต้องการจะส่งเมสเสจของวัตถุให้ทำงานตามที่เราต้องการ เช่น เกี่ยวกับการฝากเงินหรือถามยอดเงิน ใน C++ ทำได้โดยการเติมฟังก์ชันเข้าไป และฟังก์ชันนี้จะเป็นสมาชิกของคลาส เรียกว่า ฟังก์ชันสมาชิก (member function)

ฟังก์ชันสมาชิก (Member Function)

ฟังก์ชันสมาชิกจะถูกใช้ในการส่งเมสเสจและตอบรับเมสเสจจากวัตถุที่ประกาศการใช้คลาส

ตัวอย่าง

เราจะขยาย struct ของบัญชีเงินฝากจากของเดิมให้เป็นคลาสโดยเติมฟังก์ชันสมาชิกเข้าไป

```
struct account{
    double balance;
    double rate;
    void deposit (double amount);
    double withdraw (double amount);
    void compound( );
    double get__balance( );
};
```

จะเห็นว่าฟังก์ชันสมาชิกที่เพิ่มเข้าไปใน struct คือ

```
deposit( ), withdraw( ), compound( ),
get__balance( );
```

ซึ่งจะใช้จัดการกับวัตถุของข้อมูลแบบ account ฟังก์ชันที่ประกาศไว้ในคลาสบอกคอมพายเลอร์ให้รู้ว่าฟังก์ชันไหนที่สามารถใช้กับวัตถุของคลาสได้ เราใช้โอเปอเรเตอร์ . (จุด) ช่วยทำให้ฟังก์ชันสมาชิกใช้ได้กับวัตถุ (object) เช่น name1 กับ name2 เป็นวัตถุของ account การให้วัตถุเข้าถึงฟังก์ชันทำได้โดย

```
name1.deposit( );
```

ถ้อยแถลง

```
name1.deposit(1,000.0)
```

หมายถึงฝากเงิน 1,000 บาท เข้าบัญชีของ name1

```
name2.deposit(5,000.0)
```

หมายถึงฝากเงิน 5,000 บาท เข้าบัญชีของ name2

ถ้าต้องการถอนเงิน 5,000 บาทจากบัญชี name1 แล้วพิมพ์จำนวนเงินออกมาโดยจะใช้ถ้อยแถลง

```
name1.compound( );
```

คำนวณดอกเบี้ยเงินฝากจากบัญชี name1 และ

```
cout < <name2.get__balance( );
```

พิมพ์เงินปัจจุบันที่อยู่ในบัญชี name2 แต่ละถ้อยแถลงจะส่งเมจเสจไปให้แก่วัตถุ แล้ววัตถุจะส่งค่าที่ได้คืนกลับมาให้ ถึงแม้ว่าเราจะประกาศฟังก์ชันสมาชิกของ account แต่ฟังก์ชันสมาชิกนี้ก็ยังไม่ได้ถูกนิยามขึ้น จึงต้องนิยามขึ้นก่อนดังนี้

```
void account::deposit (double amount)
{
balance + = amount ;
    // balance = balance + amount
}
```

อธิบาย

account :: deposit บอกให้รู้ว่าชื่อของฟังก์ชัน deposit ได้ประกาศไว้แล้วในคลาส account (โดยใช้ :: เป็นตัวบ่งบอก เราเรียกเครื่องหมาย :: ว่าโอเปอเรเตอร์ขอบเขต ซึ่งลำดับการทำงานของโอเปอร์เรเตอร์นี้ทำจากซ้ายไปขวา) ดังถ้อยแถลงต่อไปนี้

```
name2.deposit(2,500.75);
```

คือการติดต่อกันระหว่าง name2 (วัตถุ) กับฟังก์ชันสมาชิก (ในที่นี้คือ deposit) ดังนั้นถ้าเราจะใช้ฟังก์ชันสมาชิกจำเป็นต้องมีเครื่องหมาย :: คั่นระหว่างชื่อของคลาสกับชื่อของฟังก์ชันเสมอเช่น

```
account::balance, account::rate
หรือ account::deposit
```

การเรียกใช้ฟังก์ชันสมาชิกทำได้โดยการใช้จุดคั่นระหว่างวัตถุกับฟังก์ชันสมาชิก เช่น name1.deposit(1,000.0) จะมีความหมายเหมือนกับ

```
double amount = 1,000.0
name1. balance + = amount ;
หรือการเรียกฟังก์ชัน
name2.deposit(5,000.0);
เหมือนกับ
double amount = 5,000.0 ;
name2.balance + =amount ;
```

ดังนั้นการเรียกครั้งแรกคือ name1. deposit (1,000.0) เป็นการฝากเงิน 1,000 บาทในบัญชี name1 ส่วนการเรียกครั้งที่สอง คือ name2.deposit (5,000.0) จะเป็นการฝากเงิน 5,000 บาทในบัญชี name2 เราลองมาเขียนฟังก์ชันการถอนเงินกัน ให้ชื่อฟังก์ชันว่า withdraw ( )

```
double account :: withdraw(double amount)
{
    if (amount < =balance){
        balance − =amount;
        return amount;
    }
else
    return 0.0;
}
```

ถ้าจำนวนเงินที่ต้องการจะถอนน้อยกว่าเงินในบัญชี ก็ถอนได้เท่ากับจำนวนที่ต้องการถอน มิฉะนั้นให้ค่า ϕ เช่นต้องการจะถอน 500 บาท ในบัญชีมีเงิน 700 บาท (balance) แสดงว่าถอนได้เพราะว่า

```
amount < balance
```

เมื่อถอนแล้วจะเหลือ 200 บาท ดังประโยค

```
balance − = amount;
```

แต่ถ้าเงินในบัญชีไม่พอก็จะถูกปฏิเสธ (คือถอนไม่ได้นั่นเอง) ประโยค

```
cash = name1. withdraw (500.0);
```

มีความหมายเหมือนกับถ้อยแถลง

```
double amount = 500.0;
if ( amount < = name1.balance){
    name1. balance − = amount;
    cash = amount;
}
else
    cash = 0.0;
```

ฟังก์ชัน compound( ) ซึ่งคำนวณดอกเบี้ยของเงินฝากในช่วงระยะเวลาใดเวลาหนึ่ง

```
void account :: compound ( )
{
    double interest = round ( balance *rate);
    balance + = interest;
}
```

ดังนั้นประโยค

```
name1.balance( );
```

จึงมีความหมายเหมือนกับ

```
double interest=round (name1 balance * name1.rate); name1.balance + = interest;
```

ข้อสังเกต ทั้ง balance และ rate ใช้ได้กับ name1

ฟังก์ชัน get__balance ( ) นิยามด้วย

```
double account::get__balance( )
{
    return balance;
}
```

ดังนั้นประโยค

```
cash = name1. get__balance( )
```

จึงมีความหมายเดียวกับ

```
cash = name1.balance
```

คอนสตรักเตอร์และเดสตรักเตอร์ (constructor and destructor)

เรายังไม่ได้พูดถึงการกำหนดค่าเริ่มต้นให้แก่ตัวแปรภายในคลาสเลย เช่น name1 หรือกล่าวโดยทั่ว ๆ ไปคือการสร้างค่าให้แก่ข้อมูลชนิดคลาสนั่นเอง สิ่งที่เราได้ทำไปแล้วคือการกำหนดค่าให้แก่

```
name1.balance กับ name1.rate เช่น
account name1;
name1.balance = 0.0;
name1.rate = 0.08;
```

จะเห็นว่าถ้อยแถลงข้างต้นจัดการโดยตรงเกี่ยวกับตัวแปร (instance variable) ของวัตถุ (object) เราใช้คีย์-เวิร์ด (keyword) class เพื่อส่งเสริมในการซ่อนข้อมูล แต่การทำข้างต้นผิดกฎ

การกำหนดค่าเริ่มต้นที่ใช้ใน struct ในซีและ C++ จะทำดังนี้

```
account name1 = { 0.0, 0.08 }
```

สมาชิกที่อยู่ในเครื่องหมายปีกกาจะถูกกำหนดให้แก่สมาชิกที่เป็นตัวแปรในคลาสอย่างเรียงลำดับ แบบเดียวกันกับที่กำหนดไว้ในวงเล็บเช่น

0.0 จะถูกกำหนดค่าให้แก่ name1. balance

0.08 จะถูกกำหนดค่าให้แก่ name1.rate

แต่ไม่มีการซ่อนข้อมูลเกิดขึ้น วิธีการซ่อนข้อมูลใน C++ จะใช้คอน-สตรักเตอร์คือ การใช้ฟังก์ชันช่วยในการกำหนดค่าเริ่มต้นให้แก่วัตถุ เช่น

```
struct account {
    double balance;
    double rate;
    account (double bal, double pcnt);
    void deposit (double amount);
    double withdraw (double amount);
    void compound ( );
    double get__balance( );
};
```

จะเห็นว่าไม่มีชนิดของข้อมูลใดเลยที่ฟังก์ชันคอนสตรักเตอร์ account( ) ส่งคืนกลับ เพราะว่าคอนสตรักเตอร์ account( ) จะคืนวัตถุของชนิดข้อมูลชนิด account นั่นเอง

นิยามของคอนสตรักเตอร์จะมีมากกว่านิยามของฟังก์ชันสมาชิกอื่น ๆ เช่น

```
account::account(double bal, double
pcnt)
{
    balance = bal;
    rate = pcnt/ 100.0;
```

จะเห็นว่าไม่จำเป็นต้องบอกชนิดของข้อมูลที่ส่งคืนกลับ

คอนสตรักเตอร์กำหนดค่าเริ่มต้นของวัตถุชนิด account โดยการกำหนดค่าให้แก่ตัวแปรสมาชิก และไม่ต้องมีถ้อยแถลงรีเทอร์นวัตถุที่ถูกกำหนดค่าเริ่มต้นโดยใช้คอนสตรักเตอร์จะถูกรีเทอร์นโดยอัตโนมัติ ดังนั้นการใช้คอนสตรักเตอร์ในสถานการณ์ทั่ว ๆ ไปคือ การกำหนดค่าเริ่มต้นให้แก่ตัวแปรนั่นเอง

```
account name1 = account (10000.0,
0.6);
account name2 = account (5000.0,
0.5);
```

ประโยคแรกประกาศว่า name1 เป็นวัตถุของ account ซึ่งมีเงินในบัญชี 10,000 บาทและใช้อัตราดอกเบี้ย 0.6

ประโยคที่สองประกาศว่า name2 เป็นวัตถุของ account ซึ่งมีเงินในบัญชี 5,000 บาท และใช้อัตราดอกเบี้ย 0.5

ใน C++ อาจใช้ถ้อยแถลงสั้น ๆ แทนได้เช่น

```
account name1 = account (1000.0,
0.6);
```

แทนด้วย

```
account name1(10000.0,
0.6);
```

และ

```
account name2 = account (5000.0,
0.5);
```

แทนด้วย

```
account name2(5000.0, 0.5);
```

อาร์กิวเมนต์สำหรับคอนสตรักเตอร์ให้ใส่ไว้หลังชื่อของวัตถุที่ประกาศไว้แล้วได้เลย เมื่อเราประกาศใช้คอนสตรักเตอร์สำหรับคลาส ก็คือ การกำหนดค่าเริ่มต้นของวัตถุในคลาสนั่นเอง

เดสตรักเตอร์ (Destructor) เป็นคอมพลิเมนต์ของคอนสตรักเตอร์ เมื่อต้องการจะสร้างวัตถุขึ้นมาใหม่เราจะเรียกใช้คอนสตรักเตอร์ ดังนั้นถ้าต้องการจะทำลายวัตถุเราก็ต้องเรียกใช้เดสตรักเตอร์ (Destructor) ถ้าวัตถุนั้นประกาศใช้ภายในบล็อก (และไม่ได้ประกาศว่าเป็นแบบ static) เดสตรักเตอร์จะถูกเรียกใช้เมื่อวัตถุนั้นออกนอกขอบเขต (scope) หรือพูดอีกนัยหนึ่งคือ เมื่อวัตถุนั้นกำลังจะถึงจุดสิ้นสุดของบล็อก

ชื่อของเดสตรักเตอร์ก็คือชื่อของคลาสซึ่งอยู่ต่อจากอักขระทิลดิ (tilde) หรือ สควิกเกิล (squiggle) เช่น ~account( ); คลาสหนึ่งจะมีเดสสตรักเตอร์ที่เดียว และไม่ต้องมีอาร์กิวเมนต์ เช่น

```
~ account( );
ส่วนนิยามของเดสตรักเตอร์ คือ
account :: ~ account( )
    {
        //statements for destructor
}
```

คลาสต่าง ๆ ที่ประกาศด้วยคีย์เวิร์ด class

ถ้าใช้คีย์เวิร์ด struct ตัวแปรสมาชิกและฟังก์ชันสมาชิกจะเป็นสาธารณะ (public) โดยอัตโนมัติ กล่าวคือ ทั้งคู่จะถูกเข้าถึงได้โดยใช้จุด (.) เช่น name1.balance name1.rate name1.deposit( ) และ name1.get__balance( ) ซึ่งจะเห็นว่าไม่มีการซ่อนข้อมูลเกิดขึ้น เพราะว่าโปรแกรมเมอร์สามารถเข้าถึงตัวแปรของวัตถุได้ ส่งและรับข่าวสารผ่านฟังก์ชันได้

คีย์เวิร์ด class จึงถูกนำมาใช้แทน struct เมื่อประกาศเกี่ยวกับคลาส เมื่อเราใช้ class สมาชิกของคลาสทุกตัวจะเป็นส่วนตัว (private) ไปโดยปริยาย ดังนั้นการใช้ name1.deposit (10000.0); และ cout < < name1.get__balance ( ); จึงผิดกติกา

ต่อมาจึงมีคีย์เวิร์ด private กับ public เพราะว่าสมาชิกของคลาสเป็น private โดยปริยาย ดังนั้นเราจึงไม่จำเป็นต้องมีคีย์เวิร์ด private ก็ได้ และใช้คีย์เวิร์ด public ถ้าต้องการให้โลกภายนอกเข้าถึงคลาสได้

เพื่อให้การซ่อนข้อมูลสมบูรณ์ขึ้น เราจะประกาศคลาสโดยใช้คีย์เวิร์ด class ดังนี้

```
class account{
    double balance;
    double rate;
public:
account (double bal,double pent);
void deposit (double amount);
double withdraw (double amount);
void compound( );
double get__balance( );
};
```

ด้วยการประกาศเช่นนี้ ตัวแปร instance ได้แก่ balance และ rate เป็น private ถ้าจะพยายามเข้าถึงตัวแปร balance และ rate โดยตรงจะผิดกฎทันที เช่น

```
name1.balance + = 5000.0;
```

แต่ในที่นี้เราประกาศฟังก์ชันสมาชิกเป็น public ดังนั้น

```
name1.deposit (5000.0)
```

จึงถูกกติกา

การคอมพายล์และการลิงค์

ถ้าเราต้องการใช้คลาสในโปรแกรม เราควรจะมีอินคลูด (include) ไฟล์-เฮดเดอร์ (เช่นเดียวกันกับที่เราต้องการใช้ฟังก์ชันในไลบรารี) ไว้ในแต่ละคลาสที่เราต้องการใช้ การทำเช่นนี้จะสามารถใช้คลาสได้เต็มที่ เช่นอาจใช้

– ชื่อของคลาส เพื่อประกาศและกำหนดค่าเริ่มต้นให้วัตถุ

– จัดการประกาศวัตถุให้เป็นสาธารณะ

ลองมาดูการคอมพายล์ใน C++

ในการคอมพายล์จะมีอยู่ 4-5 ขั้นตอนด้วยกัน หรือจะใช้คำสั่งเดียวที่คอมพายล์จากซอร์สโค้ดรหัสต้นฉบับไปเป็นไฟล์เอ็กซีคิวต์ (executable file) ที่เครื่องคอมพิวเตอร์เข้าใจ นั่นคือจะมีเพียง 2 ขั้นตอน คือ

ขั้นตอนแรก

ไฟล์ภาษาเครื่อง (เรียกว่าออปเจ็กต์-ไฟล์ ซึ่งออปเจ็กต์ตัวนี้ไม่เกี่ยวข้องกับออปเจ็กต์ใน oop เลย) ออปเจ็กต์ไฟล์ประกอบไปด้วยรหัสภาษาเครื่องที่เป็นตัวแทนของถ้อยแถลงที่เขียนไว้ในโปรแกรมต้นฉบับ แต่จะไม่มีรหัสสำหรับฟังก์ชันไลบรารีร่วมอยู่ด้วย

ขั้นตอนที่สอง

คือ การลิงค์ การลิงค์จะเอาออปเจ็กต์ไฟล์กับออปเจ็กต์ไฟล์จากไลบรารีรวมเข้าด้วยกันเป็นไฟล์เอ็กซีคิวต์ ซึ่งพร้อมที่จะรันได้ทันที

ในโปรแกรม C++ ประกอบด้วยไฟล์ต้นฉบับหลาย ๆ ไฟล์ แต่ละไฟล์จะคอมพายล์ให้เป็นไฟล์ออปเจ็กต์ หลังจากนั้นก็ลิงค์เข้าด้วยกัน รวมเป็นไฟล์ที่ใช้รันได้เลย (executive file) เพียงไฟล์เดียว

ลองประยุกต์การคอมพายล์ไฟล์ account การประกาศคลาสจะรวมเอาข่าวสารทั้งหลาย ซึ่งโปรแกรมต้องการใช้คลาสหรือฟังก์ชันสมาชิก ดังนั้นเราจะวางการประกาศคลาสไว้ในไฟล์-เฮดเดอร์ คือ account.h ซึ่งมีรายละเอียดดังในรูปที่ 2

นิยามของฟังก์ชัน เขียนไว้ในไฟล์ต้นฉบับ account.cpp (ซึ่งไฟล์นี้จะ include ไฟล์ account.h ไว้ด้วย) แล้วคอมพายล์ account.cpp ให้เป็นออปเจ็กต์ ดังแสดงในรูปที่ 3

สมมติว่าเราต้องการใช้คลาส account ในโปรแกรม balance.cpp จำไว้ว่าโปรแกรมนี้ต้องอินคลูดไฟล์-เฮดเดอร์ account.h ด้วย และ balance.cpp ต้องทำเป็นไฟล์ออปเจ็กต์

```
                    คอมพายล์
balance.cpp - - - - - - - > balance.obj
```

แล้วนำเอา balance.obj ไปลิงค์กับ account.obj หลังจากนั้นจึงรวมเอาไลบรารีที่จำเป็นเข้าไปด้วย ดังนี้

```
balance.obj
account.obj
library
link      เข้ากับ
balance.cpp
```

เนื่องจากการคอมพายล์และการลิงค์แตกต่างกันในแต่ละระบบคอมพิว-เตอร์ ดังนั้นควรจะปรึกษาผู้สอน คู่มือ หรือผู้เชี่ยวชาญที่อยู่ใกล้

การคอมพายล์แยกส่วนทำให้ประหยัดเวลา เนื่องจากถ้าเกิดไฟล์ใดผิดพลาดก็แก้ไขเฉพาะไฟล์นั้นแล้วคอมพายล์ใหม่ โดยไม่จำเป็นต้องคอมพายล์ทุกไฟล์ที่คอมพายล์แล้วอีก สำหรับ Turbo C++ ใช้ make ช่วยในการคอมพายล์แยกส่วน ดังแสดงในรูปที่ 4

```
// file account.h
// Header file for class account
   class account {
         double balance;
         double rate;
   public:
   account (double bal, double pcnt);
         // open new account
   void deposit (double acet); // make deposit
   double withdraw (double acct); // attempt withdraw
   void compound( ); // compound interest
   double get__balance( );
};
```

รูปที่ 2 แสดงรายละเอียดการประกาศคลาสไว้ในไฟล์เฮดเดอร์

```
account.cpp - - - - - >
// account.cpp'
// source file for class account
    #include "account.h"
         double round (double acct);
            // round to two decimal place
            // open account with starting balance bal
            //      periodic      percentage      rate      pcnt
account : : account (double bal, double pcnt)
{
         balance = bal;
         rate = pcnt / 100.0;
}
// deposit amount acct
         void account :: deposit (double acct)
         {
            balance + = acct;
         }
// attempt to withdraw amount acct;
// return amount actually withdraw
double account :: withdraw (double acct)
{
         if (acct < = balance){
            balance - = acct;
            return acct;
         }
         else
            return 0.0;
}
// Compound interest for current period and
// add to balance
void account :: compound( )
{
         double interest = round (balance * rate);
         balance + = interest;
}
// return current balance
double account :: get__balance( )
{
         return balance;
}
```

```
   // round acct to two decimal place
      double round (double acct)
            {
               long cents = long (100.0 * acct + 0.5);
               return double (cents) / 100.0;
```

รูปที่ 3 แสดงออปเจ็กต์เพื่อทำการคอมพายล์เสร็จ

```
// File balance.cpp
// Compound amount in bank account
~include <iostream.h>
~include "account.h"
   main( )
{
// get input data from user
double balance; // current balance
double deposit; // mouthly deposit
double annual__pcnt__rate;
            // annual perceutage rate
   int months; // number of months
   cout << "Starting balance:";
   cin >> deposit;
   cout << "Monthly percentage rate:";
   cin >> annual__pcnt__rate;
   cout << "Number of months:";
   cin >> mounths;
// compound new balance
   account acct (balance, annual__pcnt__rate / 12.0);
         for (int m=0;m<months;m++) {
            acct.deposit (deposit);
            acct.compound( );
         }
// Print balance
         cout << "Balance after" << months <<
         "months = $";
         cout << acct.get__balance( ) << endl;
}
```

รูปที่ 4 แสดงการใช้ make ช่วยในการคอมพายล์แยกส่วน

หมายเหตุ

ถึงแม้ว่าการคอมพายล์แยกกันจะเป็นคุณลักษณะของโปรแกรมเมอร์มืออาชีพ แต่ก็ไม่สะดวกสำหรับโปรแกรมที่ไม่ซับซ้อน หรือทำการทดลองทำแบบฝึกหัด ดังนั้นทางที่ดีเราสามารถจะนำเอาการประกาศการใช้คลาส นิยามของฟังก์ชันสมาชิก และโปรแกรมเอาไว้ด้วยกัน แล้วคอมพายล์ทุกสิ่งด้วยคำสั่งเดียว โดยเอาการประกาศคลาสไว้ตอนต้นในโปรแกรมต้นฉบับแล้วตามด้วยนิยามของฟังก์ชันสมาชิก โปรแกรมที่จะใช้คลาสจะอยู่ท้ายสุด